ALIZE - User Manual

Crédits

Le guide d'utilisation d'Alizé :

ALIZE user manual version 2.0

Last editor : Eric Charton – eric.charton [at] univ-avignon.fr

Remarque : ce document est en cours de réécriture pour prendre en compte les dernières modifications d'ALIZE dans le cadre du projet MISTRAL

Cette version reprends et enrichit la version précédente:

Toolkit version: 1.1
Auteur / Writer: Frédéric Wils – frederic.wils [at] lia.univ-avignon.fr
Last update: December 9th, 2005

English Translation: Nicolas SCHEFFER – nicolas.scheffer [at] lia.univ-avignon.fr
Last translation update: June 28, 2005

Sommaire

1. Presentation

ALIZE is a software plateform that aims to facilitate application development in the speech and speaker recognition field. ALIZE is developped at the Laboratoire d'Informatique d'Avignon (LIA) by Frédéric Wils under the direction of Jean-François Bonastre since February 2003.

Alize is composed of two distinct levels:

- A base level encapsulating the technical complexity of modules (data acquisition, computation, storage, mise à disposition de l'utilisateur des données). This level mainly prevents the user from managing memory allocation by himself.
- A higher level including utilities and algorithms manipulated by the user (list management, model initialisation, MAP algorithms, ...)

2. Base architecture

The plateform is build on top of several data and computation servers:

- Data audio server will store data that could come from either a microphone or a file or others audio sources aiming to offer the illusion of an a infinite length buffer to the user. This server is not yet implemented on the platform.
- Feature Server stores features that come either from a file; either from the result of a computation from audio data. It is also designed as virtually unlimited length buffer.
- Mixture/Distribution server aims to store speaker/speech models (Gaussian Mixture Models) computed from features or loaded from a file. There is neither a buffer notion nor a duration notion in this server.
- Statistic server regroups the most used algorithms (likelihood computation, EM...) and enables to store and to accumulate computation results to get a global mean of a set of features.

3. Tools and guidelines

ALIZE is designed on top of a ANSI C++ object ensemble, aiming to be used on the most known OSes. Modeling is done in UML. No external library is necessary except the standard library.

Base directory contains several sub-dirs, among which the most important are:

src	Plateform source code (.cpp et .h) and Makefile.am file
test	contains an application aiming to achieve the platform non-regressive tests
example1	contains platform utilization examples
man	unused since the existence of this Web page.
lib	to be used
bin	to be used

After compilation, the platform is contained in a library *libalize.a* that one have to link.

The platform has been compiled using the following environment:

Compiler	O.S
Microsoft Visual C++ 6.0	Microsoft Windows 2000
Microsoft Visual C++ 6.0	Microsoft Windows XP Professional
gcc 2.95.4	Linux
gcc 3.x	Mac G4
gcc 2.95.2	HP-UX 11.00
gcc 3.3.5 (Debian)	Linux

Programming guidelines

- Class name: the first letter is a capital, the following are in lower case excepting the first letter of each inside name. Example: *MyWellProgrammedClass*.
- Nom de fonction: same as classes, except the first letter has to be in small capital. Example: getNameOfServer()
- Non member variable class: same as for classes, except the first letter has to be in small capital. Example: *myFirstVariable*
- Member variable class: same as for variables, except non members variables has "_" as first caracter. Example: *_variableOfMyClass*

Most methods accept parameters (or returns data) in form of a reference or a constant reference. Pointers are rare.

All classes derives from the abstract *Object Class* (except the *K* class). This latter owns 2 pure virtual methods which are systematically redefined in syb-classes.

- The method *getClassName()* which returns the type (classname) of any object (*String*) ;
- The *toString()* method permits to get the object description as a character chain *String*. Useful for debug.

For a lot of classes, constructor of copy and affectation operator that have been judged has useless have not been implemented. In order to forbid them, they have been declared as private.

Each class definition is done in a file with the same name as the class, and owns the ".h" extension.

Each class implementation is done in a file with the same name as the class, and owns the ".cpp" extension.

Exception 1: to avoid a conflict between the *string.h* file from the standard Windows TM library, class files *String* are called *alizeString.h* and *alizeString.cpp*.

Exception 2: internal *K* class is declared in the file *Object.h*

Important

To avoid copies of useless objects and time cost, a lot of class methods returns a constant reference to an internal object of the class. It is the user's responsibility to make a copy of this object if he wants to modify or store it.

Always think of having a close look to the class method prototyes.

Example

```
Config c;
c.setParam("myParam", "xyz");
const String&amp; param = c.getParam("myParam"); // reference to an internal
string of c
String param2 = c.getParam("myParam"); // here, param2 is a copy of the internal
string
Config* pConfig = new Config();
pConfig->setParam("myParam", "xyz");
const String&amp; param = pConfig->getParam("myParam");
delete pConfig; // take care !! param become a reference to an inexistant String
```

4. Installation

See file *README*.

5. Platform utilization examples

Available in the sub-dir *example1*. Read file *example1/README* to know how to launch the example.

6. alize namespace

All classes as well as data types of the platform are declared inside a namespace "alize". Each class is available by putting "alize::" before it, or by declaring "using namespace alize" beforehand. It is highly recommended to prefix class name each time there is a risk of conflict with a class of the same name.

Example

```
#include "alize.h"

alize::String name("blabla");
```

ou

```
#include "alize.h"
using namespace alize;

String name("abcdefg");
```

7. Main classes

Object class

Class of all classes of the platform.

To perform debugging, two class variables (static variables) are used to count created and deleted objects. When closing an application, values of these two variables has to be equal. A diffrence indicates that some objects have not been destroyed.

Static methods *getCreationCounter()* et *getDestructionCounter()* are used to access these counters.

Characters chains

Alize owns its own class to represent strings. One can access to the C-type string (terminated by '0') thanks to the *c_str()* method, returning a pointer.

The operator << is overloaded so String can be used with the standard library iostream class.

Example

```
using namespace std;
String myString("qsdfghjklml");
cout << myString.c_str() << endl;
  ou
cout << myString << endl;
```

Exceptions

A lot of class methods might generate exceptions. To intercept these exceptions, one have to systematically round its source code by try...catch blocs.

All exceptions derive from *Exception* class.

Except *EOFException*, all exceptions must be considered has harmful and have to make the application stop. Most of the time, the exception throw will let the source object in an unstable state that is not recoverable.

See Application skeleton.

Example

```
try
{
  // appel d'une méthode de classe qui peut générer une exception
  FeatureServer fs(config, "file.prm");
  // suite du traitement normal
}
catch (Exception& e) // récupération d'une référence sur l'exception
{
  // utilisation de l'exception
  std::cout << e.toString() << std::endl;
}
```

8. Configurations

Every ALIZE-based application will use a certain amount of parameters ("features" dimension, files path, etc) that can come either from the command line or from a confguration file or declared inside the application. The *Config class is used to declare an object that will store and manage these parameters (add, change, store, load).*

- Specific methods have been implemented for a fast access:
  Example: *sampleRate = config.getParam_sampleRate()* instead of *sampleRate = config.getParam("sampleRate").*
- A inexistent parameter reading attempt will throw an exception.
- flag indicating parameter existence in the configuration (*bool existsParam_PARAMNAME*) is available without calling a method.
- It is possible to merge two configurations, i.e. copy and override parameters from one to another.

Example

```
Config config;
Config.load("myconfig.xml");          // loading
  ou
Config config("myconfig.xml");        // loading when creating the object
Config.setParam("vectSize", "32");    // parameter add
Config.setParam("loadFeatureFileFormat", "SPRO3"); // another parameter add
MixtureServer ms(config);             // create an object using a configuration
Config config2;
Config2.setParam("minLLK", "-100");
Config.setParam(config2);             // configuration merging
if (config.existsParam("minLLK"))     // test parameter existence
minLLK = config.getParam("minLLK");   // slow access
  ou
minLLK = config.getParam_minLLK();    // fast access
config2.reset();                      // delete all parameters
config.save("anOtherConfigFile.xml"); // XML save
config.save("anOtherConfigFile");     // RAW save
```

Example of a configuration file in RAW format

```
distribType             GD
vectSize                32
mixtureDistribCount     128
featureFlags            1000000
maxLLK                  100
minLLK                  -100
bigEndian               false
sampleRate              100.0
saveMixtureFileFormat   XML
loadFeatureFileFormat   SPRO3
featureServerMask       1,2,5-10
```

Example of a configuration file in XML format

```
<config version="1">
  <param name="minCov">1e-200</param>
  <param name="distribType">GD</param>
  <param name="vectSize">32</param>
```

```
  <param name="mixtureDistribCount">128</param>
  <param name="featureFlags">110000</param>
  <param name="maxLLK">100</param>
  <param name="minLLK">-100</param>
  <param name="bigEndian">false</param>
  <param name="sampleRate">100.0</param>
  <param name="saveMixtureFileFormat">XML</param>
  <param name="loadFeatureFileFormat">SPRO3</param>
  <param name="featureServerMemAlloc">10000000</param>
  <param name="featureServerMask">1,2,5-10</param>
</config>
```

Plateform parameters list

Name	Description
minCov	Variance flooring - not used yet
debug	always in configuration. Can be *true* or *false*
vectSize	Feature vectors dimension
computeLLKWithTopDistribs	complete computation "COMPLETE" or partial "PARTIAL" when computing LLK with "top" distributions
topDistribsCount	"top" distributions number. Cannot be above mixtureDistribCount
featureServerBufferSize	Feature Server Historic size (in features number)
featureServerMemAlloc	Feature Server Memory Buffer size (in octets) to read a feature file (used only by a feature Server). Do not confuse with *loadFeatureFileMemAlloc* used with file readers
featureServerMask	Used to select a subset of features in the vector, **only** used before a feature Server. Example: "1,2,5-10" to select the following parameters 1, 2, 5, 6, 7, 8, 9, 10.
featureServerMode	allows a feature vector modification read by a Feature Server (*writeFeature(...)* method. It is a necessary but unsufficient condition. In details. is set to *FEATURE_WRITABLE* or *FEATURE_UNWRITABLE*. Default value of the parameter is *FEATURE_UNWRITABLE*
featureFlags	Feature vectors flags
mixtureDistribCount	New mixture default number distribution
minLLK	Minimum value of a log likelihood in accumulators
maxLLK	Maximum value of a log likelihood in accumulators
distribType	Default mixture/distribution type. "GD" ou "GF"
bigEndian	Flag little/big endian. See the manual. Value: "true" ou "false"
sampleRate	Features sample rate. Usually 100.0 Hertz
mixtureFilesPath	Path to mixture files
saveMixtureFileFormat	Default save format: "RAW" or "XML" or "ETAT"
saveMixtureFileExtension	Default mixture file extension to save. Example: ".xml"
saveMixtureServerFileFormat	Default save format for mixture server: "RAW" or "XML"
saveMixtureServerFileExtensio	Default mixture server file extension to save. Example: ".xml"

n	
loadMixtureFileFormat	Default load format: "RAW", "XML" ou "AMIRAL"
loadMixtureFileExtension	Default mixture file extension to load. Example: ".gmm"
loadMixtureFileBigEndian	Flag little/big endian for mixture files. See the manual. Value: "true" ou "false"
featureFilesPath	Path to feature files
saveFeatureFileFormat	Default feature file extension to save. Example: "RAW", "SPRO3" ou "SPRO4"
saveFeatureFileExtension	Default feature file extension to load. Example: ".prm"
saveFeatureFileSPro3DataKind	TO use only when saving in SPRO3 foramt: acoustic feature type (FBANK, FBCEPSTRA, LPCEPSTRA, LPCOEFF, PARCOR, LAR, OTHER)
loadFeatureFileMemAlloc	Memory buffer size (in octets) to read a feature file (simple reader) or a group of files (multiple file reader). This parameter is not taken in account when *featureServerMemAlloc* is used
loadFeatureFileFormat	Default feature file reader format: "RAW", "SPRO3", "SPRO4" ou "HTK"
loadFeatureFileExtension	Default extension for feature files to read. Example: ".prm"
loadFeatureFileVectSize	Useful to read RAW feature files using a mask. Avoid conflict with *vectSize*
loadFeatureFileBigEndian	Flag little/big endian for feature files. See instructions. Value: "true" ou "false"
segServerFilesPath	path for segments/clusters files
saveSegServerFileFormat	Default segments/clusters file format to save: "RAW" ou "XML"
saveSegServerFileExtension	Default extension for segments/clusters files to save. Example: ".xml"
loadSegServerFileFormat	Default segments/clusters file format to read: "RAW"
loadSegServerFileExtension	Default extension for segments/clusters files to read. Example: ".segserv"
featureComputerNumCeps	integer - number of cepstral coefficients (default: 12)
featureComputerPreEmphasis	float - pre-emphasis coefficient (default: 0.95)
featureComputerShift	float - frame shift in ms (default: 10.0)
featureComputerLength	float - frame length in ms (default: 20.0)
featureComputerWindows	string - weighting window (default: HAMMING)
featureComputerNumFilters	integer - number of filters in the filter-bank (default: 24)
featureComputerAlpha	float - frequency warping parameter (default: 0.0)
featureComputerMel	boolean - use MEL frequency scale (default: false)
featureComputerFreqMin	float - lower frequency bound (default: 0 Hz)
featureComputerFreqMax	float - higher frequency bound (default: 0 Hz)
featureComputerFFTLength	integer - FFT length (default: 512)
featureComputerLifter	integer - liftering value (default: 0)
featureComputerEnergy	boolean - add log-energy (default: false)

Configuration check

principe

Lorsqu'on souhaite vérifier que les paramètres d'une configuration sont bien ceux attendus, on peut utiliser un *ConfigChecker* comme suit :

1. on crée un objet *ConfigChecker*
2. on lui donne la liste des paramètres à tester avec leurs caractéristiques
3. et enfin on lui demande de vérifier la configuration

En cas d'échec, une exception *ConfigCheckException* est lancée.

Méthodes pour déclarer les paramètres :

- *void addIntegerParam(const String& name, bool mandatory, bool argIsRequired, const String& desc = "")*
- *void addFloatParam(const String& name, bool mandatory, bool argIsRequired, const String& desc = "")*
- *void addBooleanParam(const String& name, bool mandatory, bool argIsRequired, const String& desc = "")*
- *void addStringParam(const String& name, bool mandatory, bool argIsRequired, const String& desc = "")*

La méthode *String getParamList()* permet d'obtenir une liste des paramètres dans une chaîne de caractères pour l'affichae de l'aide (option --help).

Exemple

```
Config c; // the config to check
c.addParam("opt1", "123");
c.addParam("opt2", "xxx");

ConfigChecker cc;
// opt1: an optional integer parameter
//       the argument is required
cc.addIntegerParam("opt1", false, true, "an optional integer parameter");
// opt2: a mandatory string parameter
//       the argument is not required
cc.addStringParam("opt2", true, false, "a mandatory string parameter");

cc.check(c);
```

If we append this piece of code to the previous example

```
std::cout << cc.getParamList() << std::endl;
```

the program will display

```
--opt1  an optional integer parameter <INTEGER>
--opt2  a mandatory string parameter <String&gt;
```

Ce qui facilite la composition du texte affiché à l'écran lors de l'utilisation de *--help*>

Voir un exemple plus complet d'utilisation de *configChecker*pour contrôler une ligne de commande.

9. Command Line

Recovering parameters from the command line is relatively easy using the *CmdLine* class. An instance is created by passing *argc* and *argv arguments*. *Method* bool displayHelpRequired() and *bool displayVersionRequired()* are used to catch a *--help* or *--version* from the command line. Method *copyIntoConfig()* can then copy command line parameters into a Config object.

In case of an error in the command line, an exception of type *Exception* is thrown.

GNU and POSIX norms are used for syntax. Each parameters can be typed in three ways:

- *-pxxx*: the character - following the name of the short option (1 character) and the paramter value (without spaces)
- *--param 123*: two character - following the name of the long option, a space and the parameter value
- *--param=123*: two character - following the name of the long option, a "=" and the parameter value

Particular case: a command *--par -1.23* will create a parameter *par* with an empty value and a parameter *1* of value *.23* ! To avoid this, one should rather use *--par=-1.23*

Example

```
int main(int argc, char* argv[])
{
  CmdLine cmdLine(argc, argv);
  if (cmdLine.displayHelpRequired())
    cout << "help..." << endl;
  else if (cmdLine.displayVersionRequired())
```

```
    cout << "Version: 1.0" << endl;
  else
  {
    Config config;
    // -> check here the parameters
    cmdLine.copyIntoConfig(config);
    cout << config.toString() << endl; // display parameters
    // usage configuration
    // and continue...
  }
}
```

Calling this program with those parameters

```
myprog -Atoto --file myfile.txt --value=-1.23
```

will display on screen

```
[ Config 0012FDB4 ]
  <A> = toto
  <file> = myfile.txt
  <value> = -1.23
```

After copying the options into a *Config* objet, a check can be donne using a *ConfigChecker* object. See an example.

10. Acoustic Models management

Alize gère les modèles acoustiques sous formes de mixtures de distributions de type "Gaussienne à matrice de covariance diagonale" ou "Gaussienne à matrice de covariance pleine".

Une mixture est un regroupement de plusieurs distributions. A chacune d'entre elles est attribué un poids. Le total des poids étant égal à 1. On ne peut pas mélanger plusieurs types de distributions dans une mixture.

Distribution

Une distribution est modélisé par la classe qui dérive de la classe générique *Distrib (*DistribGD, *DistribGF*...). La création de distribution est prise en charge par le serveur de mixtures. L'utilisateur de doit jamais détruire directement une distribution hébergée par le serveur. Sauf exception, une distribution ne doit pas être créée en dehors d'un serveur.
L'opérateur = est surchargé pour permettre de copier facilement le contenu d'une distribution dans une autre. Les opérateurs de comparaison ont également été implémentés.

Méthodes pour créer une distribution

- *DistribGD& MixtureServer::createDistribGD()*: crée une distribution de type GD. Si une distribution existe déjà dans le serveur, la nouvelle distribution aura la même dimension (vectSize). Dans le cas contraire, la dimension doit être renseignée dans la configuration (paramètre *vectSize*)
- *DistribGF& MixtureServer::createDistribGF()*: crée une distribution de type GF. Si une distribution existe déjà dans le serveur, la nouvelle distribution aura la même dimension (vectSize). Dans le cas contraire, la dimension doit être renseignée dans la configuration (paramètre *vectSize*)
- *Distrib& MixtureServer::createDistrib()*: le type de la distribution à créer doit être renseigné dans la configuration (paramètre *distribType* = "GD" ou "GF"). Si une distribution existe déjà dans le serveur, la nouvelle distribution aura la même dimension (vectSize). Dans le cas contraire, la dimension doit être renseignée dans la configuration (paramètre *vectSize*)
- *Distrib& MixtureServer::createDistrib(const Distrib::Type type, unsigned long vectSize)*

Processus de création d'une distribution GD

1. Création de la distribution
2. Mise à jour du vecteur de moyennes avec la méthode *void setMean(real_t value, unsigned long index)*
3. Mise à jour du vecteur de covariance avec la méthode *void setCov(real_t value, unsigned long index)*
4. Appel de la méthode *computeAll()* pour calculer la matrice inverse, le déterminant et une constante utilisée lors du calcul de vraisemblance. A noter: avant l'appel de la méthode, le vecteur de covariance inverse, la constante et le déterminant doivent être considérés comme invalides. Après l'appel de la méthode, le vecteur de covariance est supprimé puisqu'on a besoin que de la covariance inverse par la suite (gain de mémoire). A noter: une méthode *computeAll()* existe aussi dans la mixture. Un seul appel à cette méthode permet de mettre à jour toutes les distributions de la mixture.

Processus de création d'une distribution GF

Le processus est identique à part que le vecteur de covariance est remplacé par une matrice.

Mixture de distributions

les mixtures de distributions dérivent de la classe générique *Mixture (MixtureGD, MixtureGF*). La création d'une mixture est prise en charge par le serveur de mixtures. L'utilisateur n'a pas à détruire lui-même une mixture. L'opérateur = est surchargé pour permettre de copier facilement le contenu d'une mixture dans une autre. Les opérateurs de comparaison ont également été implémentés.

Serveur de mixtures

Ce serveur sert à stocker les mixtures et leurs distributions. Chaque mixture doit posséder une identifiant unique. Un identifiant par défaut est attribué automatiquement par le serveur. Il reste toujours possible de le changer après-coup avec la méthode *setMixtureId(const String& id)*. Une mixture est composée de plusieurs distributions. Une distribution peut être partagée par plusieurs mixtures. D'où l'utilité d'avoir un dictionnaire de distributions dans le serveur afin que chacune d'entre elle soit unique (économie de mémoire et de temps de calcul). La destruction du serveur entraîne la destruction de toutes les mixtures et des distributions. Faire un *reset()* sur un serveur est équivalent à le détruire et le recréer.

<u>Server creation:</u>

```
Config config;
MixtureServer m(config);
// use m...
```

<u>Mixture loading:</u>

```
// read a single file containing a mixture
Config c;
MixtureServer ms(c);
Mixture& m = ms.loadMixture("myMixtureFile", c);
// or if we are sure the mixture is a GD-type
MixtureGD& m = ms.loadMixtureGD("myMixtureFile", c);
// use m...
// Read a set of mixture files
Config c;
MixtureServer ms(c);
XLine l;
l.addElement("myFile1");
l.addElement("myFile2");
l.addElement("myFile3");
unsigned long indexOfTheFirstMixtureLoaded = ms.loadMixture(l, c);

// Load a full server
Config c;
// ... define parameters in c (file path, extension)
MixtureServer ms("myMixtureServerFile", c);
// or
MixtureServer ms(c);ms.load("myMixtureServerFile");
```

Mixture creation:

```
Config c;
c.setParam("mixtureDistribCount", "128"); // 128 distributions for each mixture
c.setParam("distribType", "GD"); // to deal with gaussian/diag (optional
parameter)
c.setParam("vectSize", "32");

MixtureServer ms(c);

// create a mixture with 128 distributions and vectSize = 32

Mixture& m0 = ms.createMixture(); // available only if the 3 params are declared
// or
MixtureGD& m0 = ms.createMixtureGD(); // no need param "distribType"
// or
MixtureGD& m0 = ms.createMixtureGD(128); // no need params "distribType" nor
"mixtureDistribCount"
```

Mixture saving:

```
// m is the mixture to save
m.save("myMixtureFile");
// saving a whole server
// ms is the mixture server to save
ms.save("myMixtureServerFile");
```

Mixture in XML format:

```
<MixtureGD version="1" id="mixtureId" vectSize="2" distribCount="3">
  <DistribGD i="0" weight="0.1">
    <cov i="0">1.0</cov>
    <cov i="1">2.0</cov>
    <covInv i="0">1.1</covInv>
   <covInv i="1">2.2</covInv>
    <mean i="0">3.3</mean>
    <mean i="1">4.4</mean>
  </DistribGD>
  <DistribGD i="1" weight="0.2">
    <covInv i="0">5.5</covInv>
    <covInv i="1">6.6</covInv>
    <mean i="0">7.7</mean>
    <mean i="1">8.8</mean>
  </DistribGD>
  <DistribGD i="2" weight="0.3">
    <covInv i="0">9.9</covInv>
    <covInv i="1">10.1</covInv>
    <mean i="0">11.11</mean>
    <mean i="1">12.12</mean>
  </DistribGD>
</MixtureGD>
```

Mixture Server in XML Format:

```
<MixtureServer version="1" name="serverName" vectSize="2" mixtureCount="2"
distribCount="3">
   <DistribGD i="0">
     <cov i="0">1.0</cov>
     <cov i="1">2.0</cov>
     <covInv i="0">1.1</covInv>
     <covInv i="1">2.2</covInv>
     <mean i="0">3.3</mean>
     <mean i="1">4.4</mean>
   </DistribGD>
   <DistribGD i="1">
     <covInv i="0">5.5</covInv>
     <covInv i="1">6.6</covInv>
     <mean i="0">7.7</mean>
     <mean i="1">8.8</mean>
   </DistribGD>
   <DistribGD i="2"
     <covInv i="0">9.9</covInv>
     <covInv i="1">10.1</covInv>
     <mean i="0">11.11</mean>
     <mean i="1">12.12</mean>
   </DistribGD>
   <MixtureGD id="mixtureId" distribCount="3">
     <DistribGD i="0" dictIdx="0" weight="0.1"/>
     <DistribGD i="1" dictIdx="1" weight="0.2"/>
     <DistribGD i="2" dictIdx="2" weight="0.3"/>
   </MixtureGD>
   <MixtureGD id="mixtureId2" distribCount="3">
     <DistribGD i="0" dictIdx="2" weight="0.4"/>
     <DistribGD i="1" dictIdx="2" weight="0.5"/>
     <DistribGD i="2" dictIdx="0" weight="0.6"/>
   </MixtureGD>
</MixtureServer>
```

Mixtures deletion:

-Delete a set of contiguous mixtures:

```
// ms is mixture server with 5 mixtures
// deletes mixtures #1 to #3
ms.deleteMixtures(1, 3); // Distributions are not deleted here
// deletes distributions (deletes all distributions which are not referenced by
a mixture)
ms.deleteUnusedDistribs();
```

-Delete mixtures one by one:

```
// ms is mixture server with 5 mixtures
// m0...m4 are five mixtures
// deletes all mixtures
ms.deleteMixture(m0); // Distributions are not deleted here
ms.deleteMixture(m1); // Distributions are not deleted here
ms.deleteMixture(m2); // Distributions are not deleted here
ms.deleteMixture(m3); // Distributions are not deleted here
ms.deleteMixture(m4); // Distributions are not deleted here
// deletes distributions (deletes all distributions which are not referenced by
a mixture)
ms.deleteUnusedDistribs();
```

11. Acoustic features management

Feature vectors

An acoustic feature vector is modeled by the *Feature* object. A feature object is composed of a vector of values (speech features), a valid/non-valid flag, a label code.

FeatureServer

A feature serveur is intended to provide a set of services to manage features which comme from various sources (file, set of files, microphone, ftp server...). The way to access frames is always the same: sequential access frame by frame.

A memory buffer (configurable size using parameter *featureServerMemAlloc*) can be allocated to speed up frame reading from files. The algorithm is designed for this configuration:

- numerous files;
- frequent swap between files;
- read a set of contiguous features in each file.

Le 2ème paramètre du constructeur du *FeatureServer* indique l'origine des trames:

- *FeatureServer(const Config& c, FeatureInputStream& s)*: flux externe;
- *FeatureServer(const Config& c, const FileName& f)*: un fichier de features ou un fichier audio (paramétrisation à la volée) un fichier ascii (.lst) contenant une liste de noms de fichiers de features;
- *FeatureServer(const Config& c, const XLine& l)*: une liste de fichiers dans un objet *XLine*;

Feature vectors reading

Utiliser la méthode *bool readFeature(Feature& f, unsigned long step=1)* pour lire la prochaine trame.

- *f* est l'objet qui va contenir les données lues.
- *step* indique la position de la prochaine trame à lire par rapport à la trame courante. Une valeur égale à 0 force le pointeur de lecture à rester sur place. Cela permet, par example, de faire suivre le *readFeature()* par un *writeFeature()*

La méthode renvoie *false* si la fin du fichier a été atteinte. Si elle renvoie *true*, il faut tester si la trame est déclaré invalide (*getValidity()*). Si c'est le cas, un appel de la méthode *getError()* sur le serveur permet de connaître la cause de cette invalidité (trame hors historique, trame non disponible à l'instant t, etc.).

Codes d'erreur renvoyés par *getError()*:

- FeatureInputStream::NO_ERROR = pas d'erreur
- FeatureInputStream::FEATURE_OUT_OF_HISTORY = trame hors historique

Examples

Lecture séquentielle de toutes les trames d'un fichier

```
Config c;
c.setParam("loadFeatureFileFormat", "SPRO3");
c.setParam("loadFeatureFileExtension", ".prm");
```

```
c.setParam("featureFilesPath", "/myDirectory/");
FeatureServer fs(c, "theFileName");
Feature f;
while (fs.readFeature(f))
{
  // use de f ...
}
```

Read features from a set of files

```
Config c;
c.setParam("loadFeatureFileFormat", "SPRO3");
c.setParam("loadFeatureFileExtension", ".prm");
c.setParam("featureFilesPath", "/myDirectory/");

// creation of the file list by hand
XLine list;
list.addElement("file1");
list.addElement("file2");
list.addElement("file3");

FeatureServer fs(c, list);
feature f;
while (fs.readFeature(f))
{
  // use f...
}
```

Historic

Pour revenir sur des trames déjà lues, on peut définir un historique qui va conserver ces trames. Pour une lecture sur fichiers, l'historique est par défaut de taille infinie (toutes les trames sont accessibles). Pour les autres sources (flux monodirectionnels: microphone, ftp), il est impossible de mémoriser toutes les trames et l'historique doit avoir une taille finie. On peut choisir une taille finie pour une lecture sur fichier afin de simuler un flux monodirectionnel.

Le paramètre *featureServerBufferSize* sert à fixer la taille de l'historique. S'il n'est pas présent dans la configuration, la taille de l'historique est considéré comme infinie sur un flux de fichier et nulle pour les autres flux.

Si le paramètre *featureServerBufferSize* contient "ALL_FEATURES", la taille de l'historique est considéré comme infinie sur un flux de fichier. Pour les autres, ce cas est interdit (génère une exception).

La méthode *void seekFeature(unsigned long index)* sert à se replacer sur une trame de l'historique connaissant sont numéro d'ordre.

```
Config c;
c.setParam("loadFeatureFileFormat", "SPRO3");
c.setParam("loadFeatureFileExtension", ".prm");
c.setParam("featureFilesPath", "/myDirectory/");

FeatureServer fs(c, "theFile");
feature f;
fs.readFeature(f); // read feature #0
```

```
fs.readFeature(f); // read feature #1
fs.readFeature(f); // read feature #2
fs.readFeature(f); // read feature #3

fs.seekFeature(2);
fs.readFeature(f); // re-read feature #2
```

Filtrage des paramètres acoustiques

Si on souhaite ne garder qu'une partie des paramètres acoustiques de chaque feature, il est possible de déclarer dans la configuration le paramètre *featureServerMask* avec la liste des indices des paramètres acoustiques à garder (pour annuler le masque, utiliser "NO_MASK").

```
// c is the configuration and f the file to read
//
c.setParam("featureServerMask", "1,2,5-10");
FeatureServer fs(c, f);
Feature feature;
// feature size (vectSize) will be 8. The feature will contain parameters
// 1, 2 et 5 à 10 of the original feature
fs.readFeature(feature);
```

If the feature source is an external stream, we can insert in the stream an object which type is *FeatureInputStreamModifier*. In that case the mask must be supplied directly to this object (method *setMask(const String&)*).

```
FeatureFileReader stream(c, f);            // a feature stream
FeatureInputStreamModifier filter(stream); // the filter connected to the stream
filter.setMask("1,2,5-10");                // declare the mask
FeatureServer fs(c, filter);               // the server connected to the filter

Feature f;
fs.readFeature(feature);                   // reads a feature
```

Modification des features

Dans cette version de la plateforme et avec certaines contraintes, il est possible de modifier en mémoire les features lues dans le serveur.

- Les trames doivent venir de fichiers;
- toutes les trames doivent être en mémoire: le buffer déclaré avec le paramètre *featureServerMemAlloc* doit être de taille suffisante. Taille = nombre de trames de TOUS les fichiers X taille du vecteur (vectSize) X taille d'un *float* en mémoire (4 octets);
- la paramètre *featureServerMode* doit exister dans la configuration et valoir *FEATURE_WRITABLE*;

Pour modifier une trame, on utilise la méthode *bool writeFeature(const Feature&)*. La valeur retournée indique *false* en cas d'arrivée en fin de fichier. A chaque écriture, le pointeur de trame avance d'un pas. Mise à part le fait qu'on écrit une trame au lieu de la lire, cette méthode fonctionne de la même façon que *readFeature(...)*.

```
Config c;
c.setParam("loadFeatureFileFormat", "SPRO3");
c.setParam("loadFeatureFileExtension", ".prm");
```

```
c.setParam("featureFilesPath", "/myDirectory/");
c.setParam("featureServerMask", "0-5"); // possible mask
XLine list;
list.addElement("file1");
list.addElement("file2");
list.addElement("file3");
feature f;
FeatureServer fs(c, list);

while (fs.readFeature(f, 0)) // 0 = do not move
{
  // modify f
  f[0] = 1;
  f[1] = 0.5;
  f[2] = 1.6;
  f[3] = 3;
  f[4] = 0.1;
  f[5] = 2.5;
  fs.writeFeature(f); // pay attention : only selected parameters of the mask
are modified
}
```

Après modification, les trames peuvent être sauvegardées dans un autre fichier en les relisant

```
c.setParam("saveFeatureFileFormat", "RAW");
c.setParam("featureServerMask", "NO_MASK"); // or no parameter featureServerMask
FeatureFileWriter w("./myfile.raw", c);

fs.reset(); // or fs.seekFeature(0);
while (fs.readFeature(f))
  w.writeFeature(f);
w.close(); // can be mandatory to update the file header. Automatically done if
the writer is destroyed
```

labels

Un serveur de label peut être associé à un serveur de feature (comme à n'importe quel objet dérivant de FeatureInputStream). Dans ce cas le serveur va associer à chaque feature un label contenant le nom du fichier. C'est un moyen qui permet de connaître la provenance de chaque feature lors d'une lecture de plusieurs fichiers à la suite.

```
Config c;
LabelServer labelServer;
XLine list;
list.addElement("./file1.prm");
list.addElement("./file2.prm");
list.addElement("./file3.prm");
Feature f;
FeatureServer fs(c, list,labelServer);
// use fs...
```

Paramétrisation

Un module de paramétrisation automatique inspiré de SPRO a été intégré à la plateforme.

Le fonctionnement pour un serveur des features est très simple puisqu'il suffit de déclarer dans la configuration que les fichiers sources sont des fichiers audio et non plus des fichiers de features.

Bien entendu, comme dans SPRO, il faut aussi déclarer quels sont les caractéristiques de la paramétrisation.

Exemple

```
Config c;

// specific parameters to read audio files

c.setParam("loadAudioFileBigEndian", "true"); // Optional. If undeclared, uses
param "bigEndian" instead.
c.setParam("loadAudioFileExtension", ".sph");
c.setParam("loadAudioFileChannel", 0); // Optional. Default = 0
c.setParam("audioFilesPath", "/audio/");
// no param "loadAudioFileFormat". Only format SPHERE available

// specific parameters for parameterization

c.setParam("featureComputerNumCeps", "12"); // static parameters
```

```
c.setParam("featureComputerEnergy", "true"); // energy parameter -> vectSize =
12+1 = 13

// read an audio file as if it were a feature file (automatic parameterization)

c.setParam("loadFeatureFileFormat", "SPHERE");

FeatureServer fs(c, "myAudioFile");
Feature f;

while (fs.readFeature(f))
{
  // use f
}
```

Les paramètres pour la paramétrisation sont les suivants :

- featureComputerNumCeps
  integer - number of cepstral coefficients (default: 12)
- featureComputerPreEmphasis
  float - pre-emphasis coefficient (default: 0.95)
- featureComputerShift
  float - frame shift in ms (10.0)
- featureComputerLength
  float - frame length in ms (20.0)
- featureComputerWindows
  string - weighting window (HAMMING)
- featureComputerNumFilters
  integer - number of filters in the filter-bank (24)
- featureComputerAlpha
  float - frequency warping parameter (0.0)
- featureComputerMel
  boolean - use MEL frequency scale (false)
- featureComputerFreqMin
  float - lower frequency bound (0 Hz)
- featureComputerFreqMax
  float - higher frequency bound (0 Hz)
- featureComputerFFTLength
  integer - FFT length (512)
- featureComputerLifter
  integer - liftering value (0)
- featureComputerEnergy
  boolean - add log-energy (false)

On peut tout aussi bien remplacer le nom du fichier audio par le nom d'un fichier (.lst) qui contient une liste de noms de fichiers audios ou par un objet *XLine*.

```
FeatureServer fs(c, "myListOfAudioFiles.lst");
```

ou

```
XLine list;
list.addElement("myAudioFile1");
list.addElement("myAudioFile2");
```

```
FeatureServer fs(c, list);
```

Les paramètres permettant de d'utiliser un buffer mémoire pour les features fonctionnent également dans ce cas.

12. Calculs et statistiques

Toute l'infrastructure nécessaire à la réalisation de calculs, à l'accumulation et au stockage des résultats (vraisemblance, path Viterbi, EM...) est gérée dans un serveur de statistiques *StatServer*.

Calcul de vraisemblance feature/mixture

Principe

Le but est de calculer la probabilité (vraisemblance) qu'un ensemble de trames acoustiques (des features) aient été générées par locuteur donnée (une mixture). Pour cela on va faire un calcul de vraisemblance trame par trame et calculer à la fin la moyenne des vraisemblances obtenues.

- Création d'un accumulateur de vraisemblance. Le serveur de statistique va créer et conserver l'accumulateur
- Initialisation de l'accumulateur
- Boucle de passage des trames -> calcul et accumulation de la vraisemblance
- Calcul du résultat final

```
// client = mixture
// ss = statistics server
// fs = features server
// f0...n = set of features

// build accumulator
MixtureStat& acc = ss.createAndStoreMixtureStat(client);
// reset accumulator
acc.resetLLK();
// calcul et accumulation
acc.computeAndAccumulateLLK(f0);
acc.computeAndAccumulateLLK(f1);
...
acc.computeAndAccumulateLLK(fn);
// compute log-likelihood mean
lk_t llk = acc.getMeanLLK();
```

Calcul de vraisemblance sans accumulation

```
// client = mixture
// ss = statistics server
// fs = features server
// f0...n = set of features

lk_t llk = ss.computeLLK(client, f0); // does not need accumulator
```

Calcul de vraisemblance avec les "top" distributions

Pour les mixtures qui comportent de nombreuses distributions, les temps de calculs peuvent devenir prohibitif. Il est possible de n'utiliser que les distributions les plus significatives pour le calcul de vraisemblance, les "top" distributions. Les numéros de ces distributions sont sélectionnés de la manière suivante:

- calcul des vraisemblances des distributions pour un modèle donné (modèle du monde en général). On sélectionne ce modèle lors de l'appel de la méthode *computeAndAccumulateLLK* en passant le flag *DETERMINE_TOP_DISTRIBS*
- tri des vraisemblances>
- sélection des *n* vraisemblances les plus élevées. *n* est le paramètre "topDistribsCount" de la configuration

Ensuite il suffit de préciser lors du calcul de vraisemblance avec d'autres modèles le fait qu'on utilise les "top" distributions en spécifiant le flag *USE_TOP_DISTRIBS*.

Le paramètre "computeLLKWithTopDistribs" de la configuration permet de préciser si l'on désire un calcul de vraisemblance, avec les "top" distributions, partiel (rapide mais moins précis) ou complet (moins rapide mais plus précis). Le mode "complet" corrige la vraisemblance en tenant compte des poids des distributions non utilisées.

Example

```
// config = the configuration
// world = world model
// client1 = client model 1
// client2 = client model 2
// ss = statistics server
// fs = features server
// vectSize = 1024
config.setParam("topDistribsCount", "10");
config.setParam("computeLLKWithTopDistribs", "COMPLETE"); // or "PARTIAL"
Feature f;
MixtureStat& accWorld = ss.createAndStoreMixtureStat(world);
MixtureStat& accClient1 = ss.createAndStoreMixtureStat(client1);
MixtureStat& accClient2 = ss.createAndStoreMixtureStat(client2);
while (fs.readFeature(f))
{
  accWorld.computeAndAccumulateLLK(f,DETERMINE_TOP_DISTRIBS); // determines top
distributions
  accClient1.computeAndAccumulateLLK(f, USE_TOP_DISTRIBS); // uses top
distributions
  accCLient2.computeAndAccumulateLLK(f,TOP_DISTRIBS_NO_ACTION); // uses all
distributions
}
lk_t llkClient1 = accWorld.getMeanLLK()-accClient1;
lk_t llkClient2 = accWorld.getMeanLLK()-accClient2;
```

A noter: après détermination, la liste des indices des "top" distributions est accessible avec la méthode *LKVector& StatServer::getTopDistribIndexVector()*

Maximisation de la vraisemblance avec EM

L'algorithme EM permet de modifier les caractéristiques (moyennes, covariances) d'un modèle (mixture) en présence de données (features) de façon à maximiser la vraisemblance entre ce modèle et l'ensemble des features. Le modèle de départ doit être correctement initialisé pour obtenir le meilleur résultat (initialisation à partir d'un modèle du monde par example). Lors de l'appel de la méthode *resetEM()*, le serveur de statistiques va réaliser une copie du modèle initial fournis en paramètre et c'est cette copie qui sera retournée au moment de l'appel à *getEM()*. Attention: la copie est un modèle interne au serveur de statistiques. Pour modifier ce modèle, il est nécessaire de copier le contenu dans un autre modèle (le modèle de départ par example). Cette opération se fait en utilisant simplement l'opérateur = entre deux modèles.

Example

```
// model = client model (mixture)
// ss = statistics server
// f0...fn = a set of features
// create and reset the accumulators for the model
MixtureStat& acc = ss.createAndStoreMixtureStat(model);
acc.resetEM();
// compute and accumulate data
acc.computeAndAccumulateEM(f0);
acc.computeAndAccumulateEM(f1);
...
acc.computeAndAccumulateEM(fn);
// get the result
Mixture& result = acc.getEM(); // get the internal model (cannot be modified)
  or
model = acc.getEM(); // copy the internal model
```

Calcul du chemin optimal (Viterbi)

L'objectif est de déterminer la suite d'états (mixtures) permettant de maximiser la probabilité que les données (features) correspondent à cette suite d'états.

Opérations

- création d'un objet ViterbiAccum dans le serveur de statistiques ;
- déclaration des états (mixtures) composant le HMM ;
- remplissage de la matrice des log-probabilités de transition entre états. Attention: il n'existe pas de valeurs par défaut. Il faut remplir TOUTE la matrice ;
- reset des tampons d'accumulations ;
- boucle d'accumulation des données (features) ;
- calcul et récupération du meilleur chemin (path) et de la plus forte probabilité (llp).

Example

```
// mix1, mix2, mix3 are mixtures of the mixture server
// ss is a statistic server
// f0...f2 a set of features
// build the accumulator
ViterbiAccum& va = ss.createViterbiAccum();
// build a 3-states-hmm
va.addState(mix0);
va.addState(mix1);
va.addState(mix2);

// update the log-probability transition matrix between states
va.transition(0,0) = -0.1;
va.transition(0,1) = -0.2;
va.transition(0,2) = -0.8;
va.transition(1,0) = -0.2;
...
// Reset variables of the accumulator
va.reset();
// Accumulation
va.computeAndAccumulate(f0);
va.computeAndAccumulate(f1);
...
va.computeAndAccumulate(fn);
// compute and get the optimal path and the max probability (llp)
const ULongVector& path = va.getPath();
double llp = va.getLlp();
```

Accumulateurs de trames

Un objet *FrameAccGD* ou *FrameAccGF* permet de calculer, pour un ensemble de features:

- le cumul des paramètres
- le cumul des carrés des paramètres (vecteur pour le type *GD* ou matrice pour le type *GF*)
- la moyenne des paramètres
- la covariance des paramètres (vecteur pour le type *GD* ou matrice pour le type *GF*)
- l'écart-type des paramètres (vecteur pour le type *GD* ou matrice pour le type *GF*)

Ces 2 classes dérivent de la classe de base abstraite *FrameAcc*.

Ces objets peuvent être créés en appelant les méthodes *createFrameAccGD()* et *createFrameAccGF()* d'un serveur de statistiques.

Fonctionnement:

1. Création de l'objet
2. Appel de la méthode *reset()*
3. Accumulation des trames (dans une boucle) avec la méthode *accumulate(const Feature&)*
4. Récupération des résultats avec des méthodes dédiées:
   - *const FrameAcc::DoubleVector& getMeanVect()*
   - *const FrameAcc::DoubleVector& getAccVect() const*
   - *const FrameAccGDDoubleVector& getxAccVect() const* (cumul des carrés)
   - *const FrameAccGDDoubleVector& getCovVect()*
   - *const FrameAccGD::DoubleVector& getStdVect()*
   - *const FrameAccGF::DoubleSquareMatrix& getxAccMatrix() const* (cumul des carrés)
   - *const FrameAccGF::DoubleSquareMatrix& getCovMatrix()*
   - *const FrameAccGF::DoubleSquareMatrix& getStdMatrix()*

Méthodes complémentaires :

- *unsigned long getCount() const* (nombre de trames accumulées)
- *unsigned long getVectSize() const*

13. Labels

D'ici peu, les classes *Label*, *LabelServer* et *LabelSet* risquent d'être supprimées pour cause de double emploi avec les classes *SegServer*, *Seg*... Veuillez ne plus les utiliser.

Principe

Une feature peut se voir attachée un certain nombre d'informations (nom de fichier, nom de locuteur, segment, début/fin de segment...). Ces données sont regroupées dans un objet "label".

Les labels sont souvent identiques d'une feature à l'autre. Pour éviter de multiplier les labels en mémoire, on ne crée qu'un exemplaire de chaque label sur lequel va pointer plusieurs features. Chaque label étant référencé par un numéro (code label).

L'ensemble des labels est stocké dans un "serveur de label". A chaque demande d'ajout d'un nouveau label, le serveur va vérifier qu'un exemplaire n'existe pas déjà. Si c'est le cas, il va retourner le code du label existant sinon il en crée un nouveau.

Remarques

L'utilisateur peut très bien créer ses propres labels en dérivant la classe de labels existant dans Alize.

Le code label présent dans une feature peut être utilisé indépendamment des labels et serveurs de labels.
Certains objets d'Alize créent automatiquement des labels pour les features. Example: un lecteur de fichiers de features affecte un label à chaque feature lue qui indique le nom du fichier. Ceci n'est effectif que si un serveur de label a été créé et passé en paramètre au lecteur.

Objet Label

La classe *Label* encapsule les informations suivantes:

- le label proprement dit (chaîne de caractères)
- l'origine de la feature (chaîne de caractères)
- Quatre méthodes permettent de lire et modifier ces données :

```
void setString(const String&amp;)const
String&amp; getString() const
void setSourceName(const String&amp;)
const String&amp; getSourceName() const
```

Serveur de labels

Ce serveur stocke les objets de la classe *Label* en évitant de créer des doublons.
Pour déterminer si un doublon existe, le serveur va comparer les deux informations du label proposé à l'ajout avec celles des labels déjà stockés.

A la création du serveur, un certain nombre de labels sont créés par défaut:

- code 0 = label ""(vide)
- code 1 = label "SPEAKING"
- code 2 = label "MUSIC"
- code 3 = label "MUTE"

Ces labels ne sont que des examples.

Comme le code label des features est initialisé à 0, chaque feature pointe par défaut sur le label vide.

Méthodes permettant d'ajouter, de remplacer ou de lire un label:

```
unsigned long addLabel(const Label& label, bool forceAdd = false);
void setLabel(const Label& l, unsigned long index) const;
Label& getLabel(unsigned long index) const;
```

Le paramètre *forceAdd* permet de forcer le serveur à stocker le label sans chercher à savoir s'il existe un doublon.

Méthodes pour vider le serveur ou connaître le nombre de labels stockés:

```
void clear(bool deletePreDefined = false);
unsigned long size() const;
```

Quand il est positionné à "true", le flag *deletePreDefined* permet de supprimer aussi les labels pré-définis.

Associer un label à une feature

La méthode *LabelServer::addLabel* retourne un code qui est utilisé pour faire le lien entre la feature et le label.

Dans une feature, deux méthodes permettent d'écrire et de lire le code label:

```
unsigned long getLabelCode() const;
void setLabelCode(unsigned long code);
```

Example d'utilisation

```
// soit f1, f2 des features
// création du serveur
LabelServer s
...
// création d'un label
Label l("mon label");
...
// stockage du label et affectation du code aux features
f1.setLabelCode(s.addLabel(l));
f2.setLabelCode(s.addLabel(l));
// f1.getLabelCode() retourne la même valeur que f2.getCodeLabel()
```

Fichier de labels

Un fichier de labels contient un ensemble de labels auxquels sont associés une valeur de début (en secondes) et une valeur de fin (en secondes). L'extension de ce type de fichier est *.lbl* . En mémoire, l'objet qui peut contenir ces données est une instance de la classe *LabelSet*.

Example

```
// example de fichier
// 0.0 1.5 label1
// 1.5 5.0 label2
// 10.2 18.25 label1
// création d'un objet LabelSet et chargement à partir d'un fichier
LabelSet set("labelFile.lbl", config);
// on parcoure tous les labels
for (unsigned long i=0; i < set.size(); i++)
{
  const String&amp; name = set.getName(i);
  real_t begin = set.getBegin(i);
  real_t end = set.getEnd(i);
}
```

14. Segmentation

Segment

Dans *Alize*, le terme "segment" désigne un groupe contiguë de features auquel on attache un certain nombre de caractéristiques (voir ci-dessous). Les segments peuvent être dupliqués, fusionnés ou scindés à volonté.

Un segment contient les données:

- *unsigned long begin*: index de la première feature
- *unsigned long length*: nombre de features
- *unsigned long labelCode*: code label
- *String string*: chaine de caractères (libre)
- *String sourceName*: nom de la source (libre)
- *XList list*: une *XList* pour stocker des informations diverses (libre)

Différents méthodes permettent de lire et de modifier ces données. Voir la documentation technique.

Fusion (merge) de deux segments

Les caratéristiques du 1er segment sont modifiées et le second segment est détruit.

Le début *begin* est le plus petit des deux débuts ;
La longueur *length* est la différence entre le numéro de la toute dernière feature et le début ;
Les données *string* sont concaténées si elle sont différentes ;
Les données *sourceName* sont concaténées si elle sont différentes ;
Les listes sont concaténées ;
Le code label est inchangé.

Scission (split) d'un segment à partir d'une feature donnée

Soit *i* le numéro d'ordre de la feature. Un nouveau segment est créé lors de la scission. Toutes les caractéristiques sont dupliquées sauf:

Le début *begin* du segment original est inchangé ;
La longueur *length* du segment original devient *i - length* ;
Le début du nouveau segment est égal à *i* ;
La longueur *length* du nouveau segment est recalculée à partir de *i*.

Duplication d'un segment

La méthode *Seg& duplicate()* réalise cette opération. Le segment dupliqué est identique au segment original.

Cluster de segments

On peut créer des clusters de segments (classe *SegCluster*) pour regrouper un ensemble de segments. Les clusters peuvent aussi contenir d'autres clusters (clusters hiérarchiques).
Tous les éléments d'un cluster (segments ou autres sous-clusters) peuvent être rajouté ou supprimés facilement.

Il est possible de parcourir l'ensemble des segments d'un cluster hiérarchique de façon séquentielle en faisant appel à la méthode *Seg* getSeg()* à l'intérieur d'un boucle qui se termine lorsque le pointeur renvoyé est égal à *NULL*. Avant de démarrer la boucle, penser à faire appel à *rewind()* pour positionner le pointeur de lecture sur le 1er segment.

Lorsque le cluster est entièrement constitué, on peut souhaiter savoir si une feature donnée (identifiée par son numéro d'ordre) fait partie d'un des segments du cluster. La méthode *bool getFeatureLabelCode(unsigned long n, unsigned long& lc, bool& isFirst, bool& isLast)* va parcourir l'ensemble des segments du cluster jusqu'à trouver celui qui contient la feature (début du segment >= numéro d'ordre et début du segment + longueur du segment < numéro d'ordre). Si ce segment existe, la méthode renvoit *true* et les variables *lc*, *isFirst* et *isLast* contiennent respectivement le code label du segment, un flag indiquant si la feature est la première du segment et un deuxième flag précisant si la feature est la dernière du segment (flags utiles pour le décodage Viterbi).

Nb: dans cette version de la plate-forme, le numéro d'ordre est nécessaire étant donné que celui-ci n'est pas incorporé dans la feature. Dans une prochaine version, c'est la feature qu'il faudra passer comme paramètre à la fonction à la place du numéro.

Serveur de segments

Pour faciliter la gestion des segments et des clusters, ceux-ci sont regroupés au sein d'un serveur de segments (classe *SegServer*). Lors de la destruction du serveur, tous les objets contenus dans celui-ci sont automatiquement détruits. Ainsi, l'utilisateur n'a pas à se préoccuper d'allouer ou de désallouer le mémoire.

Nb: à l'intérieur d'un serveur, chaque cluster se voit automatiquement attribué un identifiant (entier >= 0) UNIQUE qu'il est possible de changer après-coup par la méthode *SegServer::setClusterId(unsigned long id).*

Exemples d'utilisation

```
// Creation d'un serveur
SegServer ss;
// Affectation d'un nom (facultatif) au serveur
ss.setServerName("server name");

// Creation de segments
Seg& s0 = ss.createSeg(10, 20, 1, "blabla", "thefilename");
Seg& s1 = ss.createSeg(30, 2000, 1, "", "thefilename");
Seg& s2 = ss.createSeg();
s2.setBegin(2030);
s2.setLength(100);
s2.setLabelCode(1);
// ajout d'informations à un segment. Ces informations sont stockées
// dans une XList
s2.list().addLine().addElement("info1").addElement("info2");
s2.list().addLine().addElement("info3");

// Creation de clusters
SegCluster& cl0 = ss.createCluster();
SegCluster& cl1 = ss.createCluster();

// Constitution des clusters
cl0.add(s0);
cl0.add(s1);
cl1.add(s2);
cl1.addNewSeg(25, 500); // creates a new segment and adds it to the cluster
cl1.addCopy(s2); // creates a copy of s2 and adds it to the cluster
cl1.add(cl0); // un cluster dans le cluster !

// Parcours d'un cluster
cl1.rewind();
Seg* p;
while ( (p=cl1.getSeg()) != NULL)
{
  // p va successivement pointer sur s2, s0 et s1
}

// Sauvegarde du serveur
// la sauvegarde suit les règles définies au chapitre "les fichiers"
// et nécessite dans certains cas des paramètres de la configuration
```

```
// soit config un objet de type Config

// Sauvegarde au format xml avec le chemin complet et l'extension du fichier
ss.save("./filename.xml", config);
// sauvegarde au format raw sans chemin ni extension
config.setParam("segServerFilesPath", "./thePath/");
config.setParam("saveSegServerFileExtension", ".raw");
config.setParam("saveSegServerFileFormat", "RAW");
ss.save("filename", config);
```

Test de l'appartenance d'une feature à un cluster

```
SegServer ss;
Seg& s0 = ss.createSeg(10, 20, 1);
Seg& s1 = ss.createSeg(30, 2000);
SegCluster& cl = ss.createCluster();
cl.add(s0);
cl.add(s1);
unsigned long lc;
bool isFirst, isLast;

// test avec une feature d'indice 9
bool found = cl.getFeatureLabelCode(9, lc, isFirst, isLast);
// found = false: la feature n'appartient à aucun segment du cluster

// test avec une feature d'indice 10
bool found = cl.getFeatureLabelCode(10, lc, isFirst, isLast);
// found = true
// lc = 1   isFirst = true   isLast = false

// test avec une feature d'indice 11
bool found = cl.getFeatureLabelCode(11, lc, isFirst, isLast);
// found = true    lc = 1   isFirst = false  isLast = false

// test avec une feature d'indice 29
bool found = cl.getFeatureLabelCode(29, lc, isFirst, isLast);
// found = true    lc = 1    isFirst = false   isLast = true

// test avec une feature d'indice 30
bool found = cl.getFeatureLabelCode(30, lc, isFirst, isLast);
// found = true     lc = 2    isFirst = true    isLast = false
```

15. Les fichiers

Little/Big endian

Les fichiers de données peuvent contenir des données au format "little endian" ou "big endian" suivant leur provenance. C'est pourquoi il est parfois nécessaire de passer d'un format à l'autre pour pouvoir lire correctement les données.

Dans Alize, certains lecteurs de fichiers détectent automatiquement le format des données (FeatureFileReaderSPro3), d'autres n'en on pas besoin (lecteurs xml) et pour les derniers on peut décider, à la création d'un lecteur, d'imposer le mode big ou little endian ou laisser le lecteur se débrouiller avec les paramètres qu'il trouvera dans la configuration.

Remarque: à ce stade d'avancement de la plateforme, seuls les lecteurs de features et les lecteurs de mixtures sont concernés.

Example

```
Config c;

// on force le mode big endian pour 1 lecteur
FeatureFileReaderRaw r("thefile", c, NULL, BIGENDIAN_TRUE);

// on force le mode little endian pour 1 lecteur
FeatureFileReaderRaw r("thefile", c, NULL, BIGENDIAN_FALSE);

// on laisse le lecteur décider
FeatureFileReaderRaw r("thefile", c, NULL, BIGENDIAN_AUTO);
// ou
FeatureFileReaderRaw r("thefile", c);
```

Si on décide de forcer le mode à partir de la configuration, on peut:

- utiliser le paramètre général *bigEndian* qui servira pour tous les lecteurs (voir la remarque plus haut) ;
- utiliser un paramètre plus spécifique: *loadFeatureFileBigEndian* ou *loadMixtureFileBigEndian*. Dans ce paramètre est prioritaire sur le paramètre général.

Fichiers concernés:

- fichiers de features au format SPRO4, RAW, HTK ;
- fichiers de mixtures au format AMIRAL, RAW.

Lecture d'un fichier "liste"

Le chemin et l'extension du fichier doivent être indiqués.

Example

```
XList list("/data/list/myFile"); // reads the file/data/list/myFile
```

Cas particulier de la création d'un serveur de features à qui on fournit un nom de fichier "liste de fichiers de features": l'extension doit obligatoirement être ".lst" pour que le serveur ne confonde pas ce fichier avec un fichier de features.

Example

```
FeatureServer fs(config, "/data/list/myFile.lst"); // reads a list of features
files
FeatureServer fs(config, "/data/myFile.prm"); // reads one features file
```

Lecture et sauvegarde d'une configuration

Le chemin et l'extension du fichier doivent être indiqués. Si l'extension est ".xml", la configuration est lue/sauvegardée au format XML. Dans tous les autres cas, elle est lue/sauvegardée au format RAW.

Lecture d'une mixture

Si le nom du fichier commence par "/" ou par "./", le nom est utilisé tel quel. Aucun chemin ou extension n'est ajouté. Dans le cas contraire, on utilise le chemin et l'extension déclarés dans la configuration (paramètres *mixtureFilesPath*, *loadMixtureFileExtension*).

Excepté pour les fichiers dont l'extension est ".xml", il est obligatoire de renseigner le paramètre *loadMixtureFileFormat* dans la configuration.

Les lecteurs de fichiers de mixtures peuvent gérer le mode big/little endian.

Sauvegarde d'une mixture

Si le nom du fichier commence par "/" ou par "./", le nom est utilisé tel quel. Aucun chemin ou extension n'est ajouté. Dans le cas contraire, on utilise le chemin et l'extension déclarés dans la configuration (paramètres "mixtureFilesPath", "saveMixtureFileExtension"). Si l'extension du fichier est ".xml", la sauvegarde est au format XML. Dans les autres cas, le format de sauvegarde est celui déclaré avec le paramètre "saveMixtureFileFormat" de la configuration (XML, RAW, ETAT).

Sauvegarde d'un serveur de mixtures

Si le nom du fichier commence par "/" ou par "./", le nom est utilisé tel quel. Aucun chemin ou extension n'est ajouté. Dans le cas contraire, on utilise le chemin et l'extension déclarés dans la configuration (paramètres "mixtureFilesPath", "saveMixtureServerFileExtension"). Si l'extension du fichier est ".xml", la sauvegarde est au format XML. Dans les autres cas, le format de sauvegarde est celui déclaré avec le paramètre "saveMixtureServerFileFormat" de la configuration (XML, RAW).

Lecture de features

Si le nom du fichier commence par "/" ou par "./", le nom est utilisé tel quel. Aucun chemin ou extension n'est ajouté. Dans le cas contraire, on utilise le chemin et l'extension déclarés dans la configuration (paramètres *featuresFilesPath*, *loadFeaturesFileExtension*).

Les lecteurs de fichiers de features peuvent gérer le mode big/little endian.

Il est possible de définir un tampon en mémoire pour stocker les données et éviter des accès trop fréquent au(x) fichier(s). Pour cela déclarer le paramètre *loadFeatureFileMemAlloc* (valeur en

octets) dans la configuration. Chaque fois qu'un lecteur sera créé, un tampon sera alloué. Il est possible de déclarer son propre tampon sans passer par la configuration. Pour cela il faut créer un objet de type *FloatVector* (vecteur de *float*) de taille adéquate sachant qu'un *float* = 4 octets et qu'une feature nécessite un *float* par paramètre acoustique. Il suffit ensuite de passer l'adresse de ce vecteur en paramètre lors de la construction d'un lecteur.

Il existe plusieurs objets qui permettent de lire directement un fichier contenant des features:

- Un objet *FeatureFileReader* permet de lire tout fichier de feature ou liste de fichiers quel que soit son format (dans la limite des formats reconnus par Alize): RAW, SPRO3, SPRO4, HTK, liste (.lst). Il s'agit donc d'un **lecteur universel** qui évite l'utilisation des autres lecteurs. Avec ce lecteur, excepté pour les fichiers dont l'extension est ".lst", il est obligatoire de renseigner le paramètre *loadFeatureFileFormat* de la configuration ;
- Un objet *FeatureFileReaderSPRO3* permet de lire uniquement un fichier de feature au format SPRO3. ;
- Un objet *FeatureFileReaderSPRO4* permet de lire uniquement un fichier de feature au format SPRO4 ;
- Un objet *FeatureFileReaderHTK* permet de lire uniquement un fichier de feature au format HTK ;
- Un objet *FeatureMultipleFileReader* permet de lire plusieurs fichiers à la suite comme s'il s'agissait d'un fichier unique. Les fichiers doivent avoir des caractéristiques identiques (vectSize, flags et frameRate). Le format de lecture est celui déclaré avec le paramètre *loadFeaturesFileFormat* de la configuration (SPRO3, SPRO4, RAW, HTK). A noter: on peut aussi indiquer un nom de fichier qui est lui-même une liste de fichiers (attention à ne pas faire de récursivité).

La lecture séquentielle des features se fait grâce à la méthode *bool readFeature(Feature& f, unsigned long step=1).*

La méthode *void seekFeature(unsigned long idx)* permet de repositionner le pointeur de lecture de features sur une feature donnée par un index (comme la fonction *fseek* de la librairie C pour un fichier). Cela permet un accès direct à une feature sans avoir à charger toutes les features en mémoire et sans avoir besoin de parcourir séquentiellement le(s) fichier(s) depuis le début. Remarque: cette nouvelle méthode déplace juste le pointeur mais ne fait pas la lecture. Elle doit être suivie de la traditionnelle lecture séquentielle *bool readFeature(...)*. De plus, placer le pointeur en dehors du fichier ne génère pas d'erreur. Dans ce cas la lecture retournera *false*.

Examples de lecteurs mono-fichier

```
Config c;
c.setParam("loadFeatureFileBigEndian", "false");
c.setParam("loadFeatureFileFormat", "SPRO3");
c.setParam("loadFeatureFilesPath", "/thePath/");
c.setParam("loadFeatureFileExtension", ".prm");

// reads a SPRO3 file
FeatureFileReader r1("aFile", c); // format, path and extension come from the
configuration
FeatureFileReaderSPRO3 r2("aFile", c); // path and extension come from the
configuration
FeatureFileReaderSPRO3 r3("/thePath/aFile.prm", c); // does not use
configuration parameters

// gets informations
unsigned long       featureCount = r.getFeatureCount();
unsigned long       vectSize     = r.getVectSize();
const FeatureFlags& flags        = r.getFeatureFlags();
```

```
double              frameRate    = r.getFrameRate();
```

Quelques méthodes utiles pour les lecteurs de fichiers multiples:

- la méthode *unsigned long getSourceCount()* permet de savoir combien de sources sont lues par le lecteur ;
- la méthode *unsigned long getFeatureCountOfASource(unsigned long srcIdx)*: retourne le nombre de features contenu dans une des source (identifiée par un indice) ;
- la méthode *unsigned long getFeatureCountOfASource(const String& sourceName)*: retourne le nombre de features contenu dans une des source (identifiée par le nom de la source) ;
- la méthode *unsigned long getFirstFeatureIndexOfASource(unsigned long srcIdx)*: retourne le numéro de la 1ère feature d'une source (identifiée par un indice) en considérant la concaténation de toutes les sources ;
- la méthode *unsigned long getFirstFeatureIndexOfASource(const String& sourceName)*: comme pour la méthode précédente mais en identifiant la source par son nom plutôt que par son indice ;
- la méthode *const String& getNameOfASource(unsigned long srcIdx)*: retourne le nom d'une source de features (identifiée par un indice).

Remarque: pour ces méthodes, le mot "source" est à comprendre comme "fichier".

Examples de lecteurs multi-fichiers

```
Config c;
c.setParam("loadFeatureFileFormat", "HTK");
c.setParam("loadFeatureFilesPath", "/thePath/");
c.setParam("loadFeatureFileExtension", ".prm");
c.setParam("loadFeatureFileBigEndian", "false");

// reads a set of files. Forces big endian mode usage

XLine list;
list.addElement("file1").addElement("file2").addElement("file3");
FeatureFileReader r(list, c, NULL, BIGENDIAN_TRUE);
  or
FeatureMultipleFileReader r(list, c, NULL, BIGENDIAN_TRUE);

// reads a file "list of file name". Uses endian mode of configuration
FeatureFileReader r("fileNameList.lst", c);

// gets informations
unsigned long       featureCount        = r.getFeatureCount();
unsigned long       vectSize            = r.getVectSize();
const FeatureFlags& flags               = r.getFeatureFlags();
double              frameRate           = r.getFrameRate();
unsigned long       fileCount           = r.getSourceCount();
unsigned long       featureCountOfFile2 = r.getFeatureCountOfASource(2);
unsigned long       featureCountOfFile1 = r.getFeatureCountOfASource("file1");
unsigned long       index2              = r.getFirstFeatureIndexOfASource(2);
unsigned long       index1              =
r.getFirstFeatureIndexOfASource("file1");
const String&amp;       fileName            = r.getNameOfASource(2); // will
return "file3"
```

Examples d'utilisation d'un tampon mémoire

```
// auto-define buffer

Config c;
c.setParam("loadFeatureFileMemAlloc", "10000"); // 10 Ko

FeatureFileReaderHTK r(/thePath/aFile.prm", c); // uses a 10 ko-buffer

// user defines the buffer

FloatVector buff(10000, 10000); // capacity & size

FeatureFileReaderHTK r1(/thePath/aFile.prm", c, NULL, BIGENDIAN_AUTO, &buff);
// .. finished to use r1
// re-uses the buffer for another reader
FeatureFileReaderHTK r2(/thePath/anOtherFile.prm", c, NULL, BIGENDIAN_AUTO,
&buff);
```

Sauvegarde de features

Si le nom du fichier commence par "/" ou par "./", le nom est utilisé tel quel. Aucun chemin ou extension n'est ajouté. Dans le cas contraire, on utilise le chemin et l'extension déclarés dans la configuration (paramètres "featureFilesPath", "saveFeatureFileExtension"). Attention: choisir l'extension ".prm" ne permet pas à Alize de déterminer le format de sauvegarde automatiquement. Dans tous les cas, le format de sauvegarde doit être déclaré avec le paramètre "saveFeatureFileFormat" de la configuration (RAW, SPRO3, SPRO4). Pour le cas *SPRO3* il faut aussi mettre le paramètre *saveFeatureFileSPro3DataKind* dans la configuration.

Lecture d'un fichier serveur de segments/clusters

Si le nom du fichier commence par "/" ou par "./", le nom est utilisé tel quel. Aucun chemin ou extension n'est ajouté. Dans le cas contraire, on utilise le chemin et l'extension déclarés dans la configuration (paramètres *segServerFilesPath*, *loadSegServerFileExtension*). Si l'extension du fichier est ".xml", la lecture est au format *XML*. Dans les autres cas, le format de lecture est celui déclaré avec le paramètre *loadSegServerFileFormat* de la configuration (*XML*, *RAW*).

Sauvegarde d'un serveur de segments/clusters

Si le nom du fichier commence par "/" ou par "./", le nom est utilisé tel quel. Aucun chemin ou extension n'est ajouté. Dans le cas contraire, on utilise le chemin et l'extension déclarés dans la configuration (paramètres *segServerFilesPath*, *saveSegServerFileExtension*). Si l'extension du fichier est ".xml", la sauvegarde est au format *XML*. Dans les autres cas, le format de sauvegarde est celui déclaré avec le paramètre *saveSegServerFileFormat* de la configuration (*XML*, *RAW*).

16. Les X-listes

On désigne par le terme "X-liste" une instance de la classe *XList*. Une X-liste est un ensemble de lignes (*XLine*) ; chaque ligne étant un tableau d'éléments de type *String*. On peut parcourir les lignes par un index ou de façon séquentielle. Idem pour parcourir les éléments d'une ligne. Une X-liste est très utile pour récupérer de façon très simple les données d'un fichier ascii organisées en lignes de "mots". Ces fichiers sont en général des listes de noms ou de paramètres comme par example:

```
mbdb 6127 M TAR 1
mbna 6127 M TAR 25 12
mcci 6127 M TAR 0
mdjg 6136 M TAR
mefc 6136
meuv 6136 M TAR
```

Chaque ligne comporte 0, 1 ou plusieurs "mots" séparés par des espaces ou des tabulations. Les lignes vides sont ignorées. Un fichier peut être chargé dans une *XList* en spécifiant le nom du fichier comme paramètre du constructeur ou en appelant la méthode *void load(const FileName& , const Config&)*. La sauvegarde se faisant graçe à la méthode *void save(const FileName&, const Config& c)* .

Quelques méthodes de la classe *XList*

- La méthode *getAllElements()* renvoie la **totalité** des éléments de la liste dans un **seul** objet *XLine*.
- Les méthodes *getElement(unsigned long)* et *XLine::getElement()* renvoient une **référence** sur l'élément. Cela autorise la modification directe de l'élément.
- La méthode *getElements()* renvoie tous les éléments d'une ligne à partir de l'élément courant.
- La méthode *findLine(const String& key, unsigned long idx = 0)* renvoie l'adresse de la 1ère ligne dont l'élément n° *idx* est égal à *key*. Si cette ligne n'existe pas, la méthode renvoie *NULL*.

Remarque: un objet de type *XLine* peut être utilisé comme un tableau de *String*.

Pour plus de détails, se reporter à la documentation technique (Doxygen).

Example

```
using namespace alize;
Config config;
// Déclaration de la liste et chargement
XList list("mon fichier", config);
// on parcoure les lignes et les "mots" de chaque ligne
// ---------------------------------------------------
XLine* pLine;
String* pElement;
list.rewind(); // on se positionne sur la 1ère ligne
while ( (pLine = list.getLine())!= NULL)
{
  pLine->rewind(); // on se positionne sur le 1er élément
  while( (pElement = pLine->getElement()) != NULL)
  {
    // utilisation de l'élément
    (*pElement) = "ma nouvelle chaîne"; // par example, modification de
l'élément
    //...
  }
}
// on peut aussi utiliser les indices*
// ----------------------------------
for (int i=0; i<list.getLineCount(); i++)
{
  XLine& line = list.getLine(i);
  for (int j=0; j<line.getElementCount(); j++)
  {
    // utilisation de l'élément
    String&amp; element = line.getElement(j);
    //...
  }
}

// recherche dans une XList
// ------------------------

XList list;
list.addLine().addElement("0").addElement("aaaa").addELement("blabla1");
list.addLine().addElement("1").addElement("bb").addELement("blabla2");
list.addLine().addElement("2").addElement("ccccc").addELement("blabla3");
XLine* p;
// recherche la ligne dont le 1er élément vaut "1"
p = list.findLine("1"); // p pointe vers la ligne trouvée (la 2ème)
// recherche la ligne dont le 1er élément vaut "z"
p = list.findLine("z"); // p vaut NULL (rien trouvé)
// recherche la ligne dont le 2ème élément vaut "bb"
p = list.findLine("bb", 1); // p pointe vers la ligne trouvée (la 2ème)
```

17. Histogramme

La class *Histo* permet de créer facilement des histogrammes de données. Un histogramme peut être sauvegardé dans un fichier ou chargé à partir d'un fichier (format texte). Le cumul des surfaces est égal à 1.0.

L'opérateur () a été surchargé pour fournir un moyen simple de connaître le nombre de valeurs dans chaque classe (colonne).

La méthode *void div(const real_t factor)* permet d'appliquer un facteur de division à toutes les classes.

Example

```
// Création de l'histogramme avec 10 colonnes
Histo h(10);
// Enregistrement des données
h.accumulateValue(1.2);
h.accumulateValue(3.4);
h.accumulateValue(5.6);
...
// calcul de l'histogramme
h.computeHisto();
// détermination du nombre de valeurs de la classe qui contient la valeur 2.0
// double n = h(2.0);
// Sauvegarde au format raw ou GnuPlot
h.save("myHistoFile.txt"); // ou h.saveGnuplot(...)
// chargement (format raw seulement)
h.load("myHistoFile.txt");
```

18. Classes de test

Pour chaque classe de la plate-forme, une classe de test existe. L'ensemble des tests pouvant être lancé simplement en créant une instance de la classe *TestSuite* et en appelant la méthode *exec()* de cette classe. Toutes les classes de test dérivent de la classe *Test* . La méthode *exec()* doit être surchargée dans les classes dérivées et contenir les tests à exécuter.

Ces classes ne sont pas nécessaires pour utiliser la plate-forme.

Le répertoire *test/* contient une application réalisant ces tests. Il suffit de la compiler et de lancer le programme *test.exe*.

19. An application skeleton

Dans cet exemple, un fichier contient la configuration par défaut. L'application s'appelle *myprog*.

Exemples de lancement de l'application en ligne de commande

1) On ne saisit pas de nom de fichier de configuration par défaut, ce qui oblige à saisir tous les paramètres.

```
myprog --optInt=123 --optString=ABSC --optFloat=3.1415 --optBool
```

2) On ne saisi que le nom de fichier de configuration par défaut.

```
myprog --config=config.xml
```

2) On demande l'affichage de l'aide (--help).

```
myprog --help
```

Donne

```
myprog 1.0
Usage: myprog [option]...
       myprog is a speech signal processing toolkit which provides
       runtime commands implementing standard feature extraction...

--help              Show this help
--version           Show version information
--config            default config file name <String&gt;
--optInt            an integer value <INTEGER>
--optString         a string value <String&gt;
--optFloat          a float value <FLOAT>
--optBool           a boolean value <BOOLEAN>
```

3) On demande l'affichage de la version (--version).

```
myprog --version
```

Donne

```
myprog 1.0
```

Le code du squelette

```
#include <iostream> // for cout
#include "alize.h"
using namespace std;
using namespace alize;
#define MANDATORY true
#define OPTIONAL false
#define ARG_REQUIRED true
#define ARG_IS_OPTIONAL false

int main(int argc, char* argv[])
{
```

```
  using namespace std;
  using namespace alize;

  try
  {
    // --------------------------------
    // Declares options for command line
    // --------------------------------

    ConfigChecker cc;
    cc.addStringParam("config", OPTIONAL, ARG_REQUIRED, "default config file
name ");
    cc.addIntegerParam("optInt", MANDATORY, ARG_REQUIRED, "an integer value");
    cc.addStringParam("optString", MANDATORY, ARG_IS_OPTIONAL, "a string
value");
    cc.addFloatParam("optFloat", MANDATORY, ARG_REQUIRED, "a float value");
    cc.addBooleanParam("optBool", MANDATORY, ARG_REQUIRED, "a boolean value");
    // ...

    // ---------------------------
    // Gets options on command line
    // ---------------------------

    CmdLine cmdLine(argc, argv);

    // -----------------------
    // Deals with option --help
    // -----------------------

    if (cmdLine.displayHelpRequired())
    {
      cout << "myprog 1.0" << endl
           << "Usage: myprog [option]..." << endl << endl
           << "       myprog is a speech signal processing toolkit which
provides" << endl
           << "       runtime commands implementing standard feature
extraction..." << endl << endl
           << cc.getParamList();
       return 0;
    }

    // --------------------------
    // Deals with option --version
    // --------------------------

    if (cmdLine.displayVersionRequired())
    {
      cout << "myprog\n 1.0" << endl;
      return 0;
    }

    // -----------------------------
    // Looks for default configuration
    // -----------------------------
```

```
    Config tmp;
    cmdLine.copyIntoConfig(tmp);
    Config config;
    if (tmp.existsParam("config")) // --config="aConfigfile.xml"
      config.load(tmp.getParam("config"));

    // ------------------------------------------------------------------
    // writes or overwrites config parameters with command line parameters
    // ------------------------------------------------------------------

    cmdLine.copyIntoConfig(config);
    cc.check(config); // throws a ConfigCheckException exception if an error
occurs
    if (config.getParam_debug())
      cout << "mode debug" << endl;

    // ------------
    // performs job
    // ------------

    // the job...

  }
  catch (ConfigCheckException& e)
  {
    cout << e.msg << endl
         << "Try test --help for more informations" << endl;
    return -1;
  }
  catch (Exception& e)
  {
    cout << e.toString() << endl;
    return -1;
  }
  return 0;
}
```